UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Diseño e implementación de un monedero para criptomoneda Bitcoin

Autor: Jiawei Tang
Tutor: Óscar Delgado Mohatar
Ponente: Álvaro Ortigosa Juárez

Febrero 2019

Diseño e implementación de un monedero para criptomoneda Bitcoin

AUTOR: Jiawei Tang
TUTOR: Álvaro Ortigosa
PONENTE: Oscar Delgado

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Febrero de 2019
Código de asignación: 1718_075_IS

Figura 1. Logo de la aplicación

Resumen

Blockchain es una tecnológica que se ha puesto de moda en estos últimos años, uno de los productos creados por esta tecnología es Bitcoin, el motivo principal de este TFG es desarrollar un monedero de Bitcoin y desde este punto enseñar al lector como funciona Bitcoin y los conocimientos que tiene que saber para desarrollar un monedero Bitcoin.

Este TFG concretamente es desarrollar un monedero Bitcoin que cumple los requisitos básicos de un monedero: crear un monedero nuevo, enviar Bitcoin, recibir Bitcoin y mantener Bitcoin de forma segura.

Este documento describe el diseño y la implementación de este sistema, usando patrones de diseño MVC junto con Singleton, en la parte de desarrollo se ha establecido el uso de lenguaje de programación TypeScript, acompañado con frameworks: Angular6, Ionic3 y Electron, y algunas librerías de NodeJs: BitcoinJs-lib y Crypto, para desarrollar la lógica interna del monedero de Bitcoin. El resultado final es obtener un monedero de Bitcoin diseñado como una aplicación hibrida, que es capaz de ejecutar en todas las plataformas ya sea dispositivo móvil o computadores, en tiempo real.

Palabras clave

Bitcoin, monedero, Blockchain, Angular6, Ionic4, Electron, BitcoinJs-lib, Crypto, Aplicación hibrida.

Abstract

Blockchain is a technology that has become fashionable in recent years, one of the products created by this technology is Bitcoin, the main reason for this thesis is to develop a Bitcoin wallet, from this point teach the reader how Bitcoin works and the knowledge the lector needs to know to develop a Bitcoin wallet.

This thesis specifically is to develop a Bitcoin HD wallet, the basic requirements of Bitcoin HD wallet are: create a new wallet, send Bitcoin, receive Bitcoin and keep Bitcoin safe.

This document describes the design and implementation of this system, using MVC design patterns together with Singleton patterns, in the development part, this thesis use TypeScript as main programming language, accompanied frameworks: Angular6, Ionic3 and Electron, and some libraries of NodeJs: BitcoinJs-lib and Crypto, to develop the internal logic of the Bitcoin HD wallet. The final result is a Bitcoin wallet designed as a hybrid application, capable of running on all platforms, mobile device or computers, in real time.

Keywords

Bitcoin, wallet, Blockchain, Angular6, Ionic4, Electron, BitcoinJs-lib, Crypto, Hybrid application.

Agradecimiento

A mi tutor de la carrera Simone Santini por haber ayudado tanto y tanto en toda la carrera, le agradezco de verdad.

A mi tutor de TFG, Oscar y Alvaro Ortigosa por darme la oportunidad de realizar este TFG con ellos.

A mis amigos Ángel y Manuel que me ha ayudado bastante durante la realización de este TFG, y también a mi amigo Jinle.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE EJEMPLOS

INDICE DE URL

1 Introducción

1.1 Motivación

Uno de los factores más importantes en realizar este trabajo, es la preocupación en por dónde y cómo guardar Bitcoins, en un lugar seguro y cómodo. Podemos usar los monederos que ya existen en el mercado o implementar nuestro propio monedero.

Hemos adoptado a la segunda opción, la implementación de nuestro propio monedero añadiendo algunos requisitos extras que cómoda al usuario final. El usuario final puede usar la misma aplicación en todas las plataformas y todos los datos del usuario están controlados por él mismo sin depender de un tercero.

1.2 Objetivos

El objetivo principal de este trabajo es realizar un monedero de Bitcoin Ligero determinista, este monedero será capaz de recibir, guardar y enviar Bitcoins del usuario. Nos enfocaremos sobre todo en tres aspectos, que consideramos esenciales para conseguir un sistema útil que satisface las necesidades de un gran número de usuarios:

- Diseño de un interfaz capaz de ser ejecutado en todas las plataformas: Windows, Linux, iOS y Android.
- Desarrollo de un código eficiente y seguro, con la capacidad de ejecución en tiempo real.
- Entender qué es Bitcoin y cómo guardar los Bitcoins de forma segura.

1.3 Organización de la memoria

La memoria está dividida en seis partes: introducción, estado de arte, diseño, desarrollo, pruebas y conclusión

En las primeras dos partes (introducción y estado de arte), se ofrece una visión del contexto en que se enmarca este trabajo, pasando por una explicación básica de en qué consiste Bitcoin, los tipos de monederos que hay, los estándares que sigue cada monedero y el objetivo final de este TFG.

En la parte de diseño se hace un repaso de las decisiones que se han tomado a más alto nivel, las tecnologías que se han usado y la arquitectura que sigue este proyecto. En la parte de desarrollo, se explican las librerías, los métodos y se dan algunos ejemplos de implementación del monedero. En la parte de pruebas, se incluyen capturas y pruebas realizadas de la aplicación. Finalmente, en la parte de conclusiones y trabajo de futuro, se hace una reflexión del trabajo realizado y se exponen posibles mejoras.

1.4 Bitcoin

En este capítulo se centra en explicar en qué consiste Bitcoin, la lógica que sigue y los componentes que forman parte de este elemento. No se trata de una explicación extensa de todo el sistema Bitcoin sino simplemente una breve explicación para entender su funcionamiento.

1.4.1 ¿Qué es Bitcoin?

Bitcoin es un conjunto tecnologías que conforman un ecosistema de dinero digital. Lo que llamamos Bitcoin consiste en datos almacenados en una red P2P. Estos datos poseen un uso financiero, es decir, tiene un uso análogo a cualquier de las divisas nacionales que se usan normalmente en el mercado (EURO, USD, YUAN…). El usuario que tenga Bitcoin puede usarlo como una divisa normal: realizar una compra venta, enviar dinero a personas y organizaciones, etc..

A diferencia de las monedas tradicionales, los Bitcoins son completamente virtuales, no existen monedas físicas. En el sistema monetario tradicional, existe unas monedas físicas: billetes de 50 euros, 20 euros, 10 euros, 5 euros y monedas de 1, 2, 0.5…. usamos estas monedas físicas para realizar movimientos financieros, para demostrar la propiedad de estas monedas, el usuario debe tener estas monedas físicas en su monedero físico o estar depositadas en un sistema centralizado. En sistema Bitcoin no existe el concepto de monedas físicas, la moneda Bitcoin está implícita en las transacciones que mueven valores de un remitente a un destinatario. Para demostrar la propiedad de los Bitcoins el usuario posee una clave única que permite realizar una transferencia de unos datos a otro usuario (Imaginase una cuenta bancaria; el usuario posee un PIN, y cada vez que el usuario quiere realizar una transferencia debe firmar la transacción con el PIN. En este contexto PIN es como la clave para firmar la transacción en la red Bitcoin. Esta clave suele estar almacenadas en una cartera digital denominado monedero Bitcoin).

En la red Bitcoin no hay un servidor ni hay un centro de control, sino que sigue un sistema P2P distribuido. Los Bitcoins se crean mediante un proceso llamado “minería” que se basa en una competencia por encontrar soluciones a problemas matemáticas a la vez que se procesan transacciones Bitcoin. Todo el mundo puede ser minero, aportando computación a la red Bitcoin, con este poder de computación cada 10 minutos puede confirmar y registrar nuevas transacciones en la red Bitcoin. El minero que consiga a resolver el problema es recompensado por nuevos Bitcoins, esta recompensa proviene de una transacción especial llamada “coinbase”, está transacción no mueve Bitcoins sino que los crea de nuevo. El protocolo Bitcoin incluye algoritmos que regulan la función de minería de toda la red, la dificultad del problema matemático mencionado anteriormente se ajusta dinámicamente, para ser resuelto en un promedio de 10 minutos, por otro lado, también regula la producción de Bitcoin (coinbase), reduciéndose la producción a la mitad cada 4 años.

Al contrario de las monedas físicas, la cantidad de Bitcoin en existencia no es potencialmente infinita, el número total de Bitcoin que se pueden crear está fijada en 21 millones. La cantidad de Bitcoin existente actualmente está por debajo de este límite, nuevos Bitcoins serán producidos cada 10 minutos hasta el año 2140, cuando la cantidad de Bitcoin en circulación llegará techo final de 21 millones.

1.4.2 Conceptos y funcionamiento de Bitcoin

En esta sección vamos a explicar los componentes básicos que hay en el sistema Bitcoin.

1.4.3 Clave Privada, Clave Pública, Dirección

Uno de los elementos más importantes en el sistema Bitcoin es la clave privada, en ella se va a derivar la clave pública luego la dirección. La existencia de la clave privada es para demostrar la pertenencia de los Bitcoins, en otras palabras, firmar una transacción.

La clave privada es un número binario de 256 bits, y la generación de esta es totalmente aleatoria. Pongamos un pequeño ejemplo: el usuario puede generar su clave privada sin depender de ninguna aplicación. Coja una moneda, un papel y un bolígrafo, apunte la cara de la moneda como 0 y la otra cara como 1, tire la moneda 256 veces, cada vez que tire la moneda, apunte el resultado de la salida, al final va a tener un resultado binario aleatorio de 256 bits, esta será tu clave privada.

Acordarse los 256 bits es bastante costoso, para simplificar este número es posible cambiar a formato hexadecimal de 32 dígitos cada uno con 4 bits. Si quieres que tu clave privada sea reconocida por los monederos de Bitcoin que hay en el mercado, debes codificar los datos con Base58 y añadir el Base58 checksum correspondiente, el formato después de cambiar a Base58 es lo que se denomina: WIF.

En las aplicaciones del mundo real usan generadores de números aleatorios de alto rendimiento para generar la clave privada, en este TFG hemos optado uno los generadores más usados: CSPRNG. La generación de números aleatorios en un CSPRNG usa entropía obtenida de una fuente de alta calidad, que puede ser un generador de números pseudoaleatorios en hardware o incluso procesos de sistemas impredecibles.

Una vez generada la clave privada se debe derivar de esta, la clave pública. Para la generación de la clave pública se tiene que adoptar el uso de ECC, el resultado obtenido por la función ECC a partir de la clave privada, es un resultado irreversible y la duración del proceso es mucho más rápido que los métodos antiguos como RSA.

Una vez obtenida la clave pública, es necesario aplicar dos funciones hash sobre ella. En el protocolo de Bitcoin han optado el uso de SHA256 y RIPMD160, este proceso es necesario para generar luego las direcciones.

$$HASH\_CLAVE\_PUBLICA = RIPMD160(SHA256(CLAVE\_PUBLICA))$$

Este doble hash tiene otro nombre más fácil y simplifica la escritura: Hash160, quedaría la nueva función de la siguiente forma:

$$HASH\_CLAVE\_PUBLICA = Hash160(CLAVE\_PUBLICA)$$

Para obtener la dirección del Bitcoin se debe codificar el hash de la clave pública con Base58 y añadir Base58 checksum correspondiente.

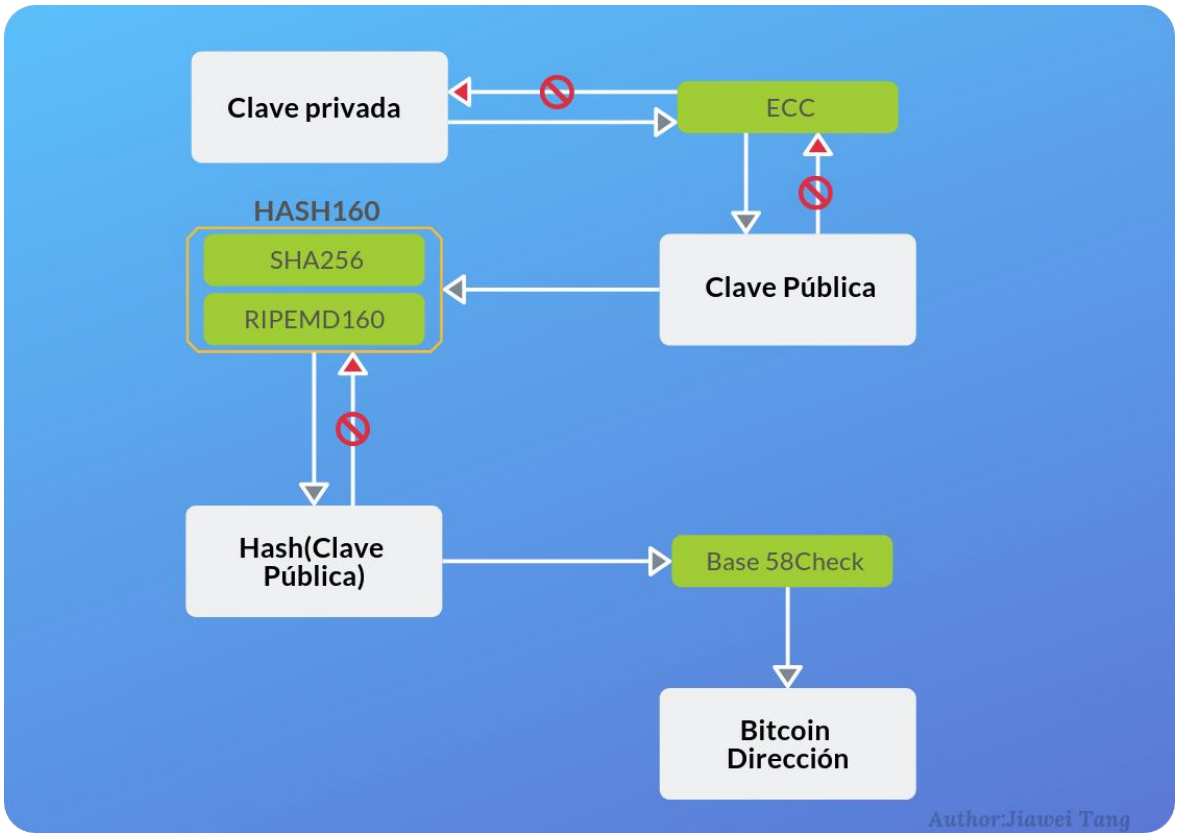


Figura 2.Clave privada - dirección Bitcoin

1.4.4 Transacción

La transacción es uno de los factores más importante en el funcionamiento de Bitcoin, con ella se realiza todos los movimientos financieros. En esta sección se explican los componentes que hay dentro de una transacción.

1.4.4.1 Blockchain

Blockchain, conocida como cadena de bloques, se trata de una estructura de datos que la información contenida se agrupa en conjuntos, denomina bloques. Cada bloque incluye metainformaciones del bloque anterior, esto hace que el sistema tenga una forma similar a una cadena. Gracias a esta estructura la información contenida de un bloque solo puede ser repudiada o editada modificando todos los bloques posteriores.

Este sistema es usado en la red de Bitcoin para almacenar y realizar transacciones. La cadena está formada por bloques, cada bloque contiene informaciones de las transacciones y una cabecera, dentro de la cabecera hay un identificador que es generado por el algoritmo SHA256, y otro hash del bloque padre (el bloque anterior). La secuencia de los hashes que unen cada bloque a su padre crea una cadena que se remonta hasta el final del primer bloque, este es conocido como bloque génesis.

1.4.4.2 Input, Output, Fee

Todo Bitcoin proviene a bien del bloque en que fue minado (coinbase) o de un movimiento de Bitcoin (una transacción). Los nodos de Bitcoin mantienen un registro de todas las salidas de transacciones que están aún por gastar, son los denominados UTXO (Unspent Transaction Output), en fin, los UTXO son Bitcoins no gastadas y están bloqueadas con un script. El propietario de esos Bitcoin necesita usar su clave privada para desbloquear esos Bitcoins (concretamente es calcular el hash de la clave privada, y que este hash debe coincidir con el que hay en UTXO), pongamos un pequeño ejemplo sobre UTXO. Un usuario A tiene 0.5 Bitcoins y realiza una transacción T que contiene 0.5 Bitcoins, al usuario B, el usuario A debe usar su clave privada para desbloquear los 0.5 Bitcoins contenidos en UTXO y luego ser enviados. En un bloque se registrará la transacción T, luego esos 0.5 Bitcoins ha de ser trasladados del UTXO del usuario A al usuario B.

UTXO no es divisible, si el usuario A tiene 1 Bitcoin en su UTXO y quiere enviar 0.3 Bitcoins al usuario B, el usuario A no tiene más remedio de enviar 1 Bitcoin entero, pero con dos direcciones, la primera dirección con un valor de 0.3 Bitcoins al usuario B, y la segunda dirección que indica la vuelta que recibe el usuario A, en este caso 0.7 Bitcoins, a la dirección del usuario A. Existen anotaciones específicas para simplificar la explicación de este proceso, elementos llamados: Input y Output.

Con el mismo ejemplo del párrafo anterior, y con anotaciones de Input y Output, si un usuario A que contiene 1 Bitcoin en su UTXO y quiere enviar 0.3 Bitcoins al usuario B. El usuario A crea un input con 1 Bitcoin (fíjense que los inputs son UTXO), y luego dos outputs, el primer output con 0.3 Bitcoins para el usuario B y el segundo output con 0.7 Bitcoins de vuelta al usuario A. Estos dos outputs ahora se convierten otra vez en UTXO,

es decir un output no gastados. Una transacción puede contener no solo un input o un output, puede contener varios inputs si los Bitcoins son recibidos por varios outputs, y también contener varios outputs en caso de que el usuario quiere mandar sus Bitcoins a varias personas en una sola transacción.

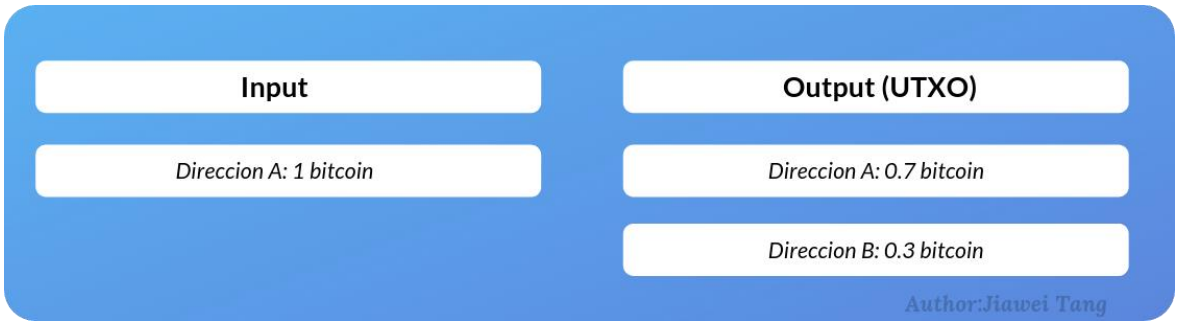


Figura 3.Input, Output, sin Fee

En una transacción no solo existe el concepto de input y output, sino también un elemento llamado fee. El lector puede tratar este elemento como una paga extra que se utiliza para incentivar a los mineros a recoger transacciones enviadas. El fee se obtiene por la diferencia de input y output, si ahora el usuario A quiere enviar 0.3 Bitcoins al usuario B con un fee de 0.1 Bitcoins, el usuario A debe crear un input de 1 Bitcoins y dos output, uno de 0.3 Bitcoins correspondiente al usuario B, y el otro corresponde la vuelta del usuario A, en este caso 0.6 Bitcoins, fíjense que la suma de outputs son 0.9 Bitcoin e input es 1 Bitcoin, con una diferencia de 0.1 Bitcoins que corresponde la fee.

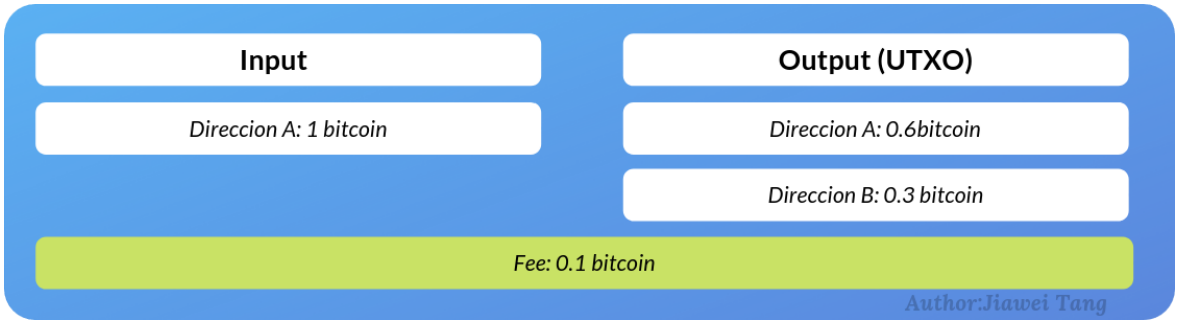


Figura 4.Input, Output, con Fee

2 Estado del arte

En este capítulo, describimos el estado de arte de los monederos de Bitcoin, haciendo referencia a los tipos de monederos que ya existen en el mercado. En el mismo capítulo explicamos los estándares que hay en el sistema Bitcoin.

2.1 Monedero Bitcoin

Un monedero de Bitcoin es el espacio virtual en que se almacenan las claves necesarias para efectuar operaciones con Bitcoin.

Existen tres tipos de monedero:

- Monedero Full Node: consiste en un monedero de Bitcoin que mantiene toda la información de Blockchain en el cliente. Este tipo de monedero usa mucha memoria de disco duro (aproximadamente ~15GB), y hay que mantener todas las informaciones de Blockchain renovadas para su correcto funcionamiento.

- Monedero ligero SPV: Simplied Payment Verification Nodes, consiste en un monedero de Bitcoin que mantiene toda la información de las cabeceras de los bloques de Blockchain en el cliente. Este tipo de monedero es ligero comparado con los monederos Full Node (aproximadamente ~200Mb), también hay que mantener todas las informaciones de las cabeceras de los bloques de Blockchain renovadas para su correcto funcionamiento.

- Monedero ligero vía API: consiste en un monedero de Bitcoin que solo mantiene las informaciones necesarias (clave privada) para firmar las transacciones, las informaciones de Blockchain se obtiene preguntando a los nodos que se encuentra en algún servidor. Este tipo de monedero no ocupa mucha memoria de disco duro(~20MB).

El objetivo final de este TFG es desarrollar un monedero ligero vía API que se puede ejecutar en todas las plataformas. Hoy en día, la mayoría de los monederos que existen en el mercado son de tipo. Entre los más conocidos son los monederos Electrum y Exodus. Su principal limitación es la imposibilidad de usarlos en dispositivos móviles. El objetivo final de este TFG es mejorar este factor y que el usuario final pueda confiar en una sola aplicación y usarla en cualquier plataforma. Escribimos el código una vez de tal forma que sea posible ejecutarlo en múltiples plataformas.

2.2 BIP

El protocolo Bitcoin no es inmutable, sino que sus funcionalidades y sus características pueden ser modificadas o amplificadas. Para ello hay que seguir unos procesos, denominados BIP (Bitcoin Improvement Proposal). En fin, BIP no son más que estándares establecidas para llevar al consenso de la comunidad de Bitcoin. En este TFG va a seguir algunos estándares establecidos: BIP0032, BIP0044, BIP0039.

2.2.1 BIP0039

BIP0039 es un estándar para facilitar la memorización de la clave privada. Es difícil de acordar la clave privada ya que es bastante larga y confusa. Esta propuesta consiste en codificar la clave privada en palabras de cualquier país, la longitud de palabras es determinada por el número de bits que forman, la clave privada suele ser de 128 bit más 4 bit de checksum que suman 132bit y estos son codificados en 12 palabras.

2.2.2 BIP0032

BIP0032 uno de los estándares más importantes, consiste en desarrollar una cartera de Bitcoin determinista (Hierarchical deterministic wallet or HD walled defined), lo que quiere decir, es que puede generar muchas claves privadas con una sola “semilla”, puede imaginarse un árbol de claves privadas y que la raíz del árbol es la semilla o la clave privada maestra y de ella se puede derivar las claves hijas y nietas. Todas las claves privadas generadas pueden producir una clave pública luego una dirección, esto nos ayuda a tener muchas direcciones de Bitcoin con una sola clave privada maestra o semilla, es bastante importante que el usuario guarde la semilla en un lugar seguro, ya que la perdida de esta semilla significa la perdida de todas las claves hijas generadas por ella.

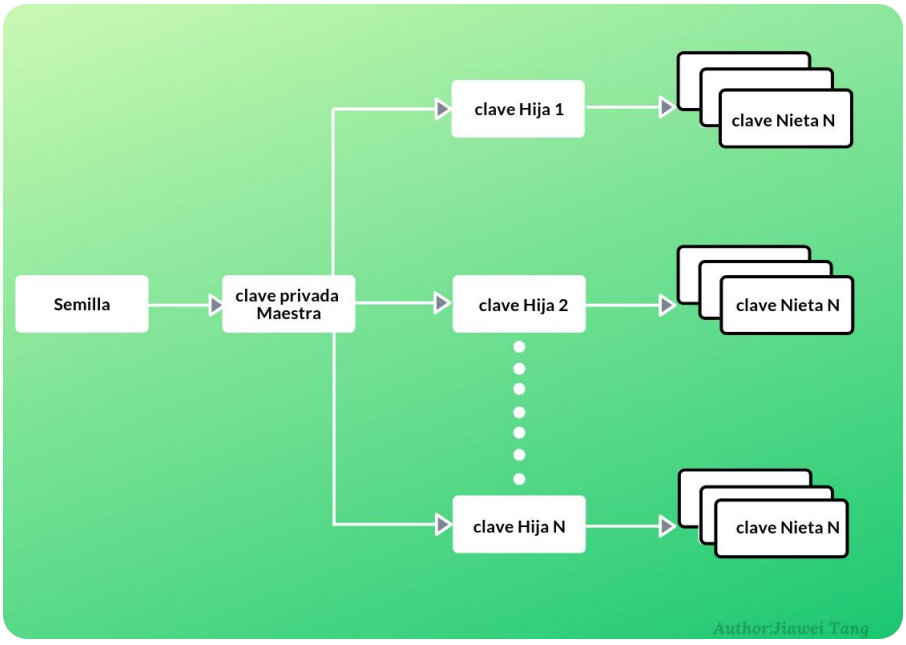


Figura 5. Estructura de BIP0032

2.2.3 BIP0044

BIP0044 es un estándar extendido del BIP0032, en BIP0032 explicamos, desde una clave privada maestra podemos derivar muchísimas claves privadas hijas. Esto conlleva a la pregunta de ¿Qué clave privada usar de todas? para tener un consenso entre todos los monederos HD, BIP0044 propuso que todos los monederos HD se deben usar la clave privada de la rama 44 del árbol de la clave privada. Hay una anotación específica para recorrer el árbol de la clave privada denominada "rutas", esta anotación tiene que ser un estándar para todos los monederos HD.

M/Purpose/Coin_type/Account/Change/Addresss_index

- M: es un costante.
- Purpose: es un constante siempre tiene que ser 44.
- Coin_type: puede ser de 3 tipos,
  - 0 indica que la clave privada pertenece a la red principal de Bitcoin
  - 1 indica que la clave privada pertenece a la red test de Bitcoin
  - 2 indica que la clave privada pertenece a la red de litecoin
- Account: indica la cuenta, que permite a los usuarios a subdividir su cartera. Pongamos un pequeño ejemplo, un usuario puede tener más de una cuenta, la primera quiere usar para pagar el crédito del coche, el segundo para alquiler de la casa, en este caso tiene dos cuentas. Podemos indicar en el primer caso como un 0 y el segundo caso un 1. Cada cuenta es la raíz de su propio subárbol.
- Change: es aquí donde se genera las direcciones de Bitcoin, nótense que en un monedero HD tiene dos subárboles, el primero tiene las direcciones de recepción y el segundo direcciones de cambio.
- Address_index: Las direcciones ya están generadas en la capa anterior "Change", hay que indicar que dirección utilizar, para eso está Address_index como el índice de dirección.

Ruta HD	Significado
M/44/0/0/0/0	La primera dirección receptora para la primera cuenta, de la red principal de Bitcoin
M/44/1/8/0/3	La cuarta dirección receptora para la novena cuenta, de la red test de Bitcoin
M/44/0/2/1/1	La segunda dirección de cambio para la tercera cuenta, de la red principal de Bitcoin

Tabla 1.Ejemplo de la enrutación BIP0044

3 Diseño

3.1 Introducción

En el apartado de diseño, está enfocado en describir todas las tecnologías, las decisiones tomadas, los patrones de diseño usado y cómo es la aplicación final.

3.2 Arquitectura

En esta sección explicamos las diferentes tecnologías que hemos planeado a usar. Concretamente son Frameworks que nos ayudan a facilitar el desarrollo del proyecto, el objetivo final es realizar una aplicación híbrida que sea capaz de ejecutarse en todas las plataformas.

3.2.1 Angular6

Es un Framework para aplicaciones web, desarrollado en TypeScript. Este Framework ayuda a estructurar el proyecto de forma modularizada con una simulación al patrón de diseño MVC (Modelo, Vista, Controlador), y de esta forma más fácil de realizar los test en sus módulos.

En los proyectos Angular se puede dividir en tres partes, los componentes (vista), los servidores (controlador), tipo de datos recibidos (modelo). Los componentes se encargan de la visualización de la aplicación que corresponde con la parte de la interfaz del usuario, los proveedores se encargan de implementar toda la lógica que hay por debajo de la interfaz del usuario, y el modelo son definiciones y declaraciones de tipos de datos recibidos y usados por el servidor.

Hemos dividido nuestro proyecto en tres partes, primero son los componentes, que sirven para decorar y estructurar la interfaz del usuario, usaremos HTML5 y SCSS, los servidores que contienen la lógica de tratar los datos y por último, la declaración de los tipos de datos que existen en nuestro proyecto.

3.2.2 Ionic4

Ionic es un Framework implementado por encima de Angular y de Apache Cordova. Su objetivo principal es crear una aplicación web que sea capaz de ejecutar en todos los dispositivos móviles, sea Android o IOS.

Usamos Ionic Framework para desarrollar este TFG. Gracias a su API a través del Apache Cordova es posible acceder fácilmente todos los componentes que hay en un dispositivo móvil, así obteniendo una aplicación web desarrollada con Angular y que es capaz de ejecutar en un dispositivo móvil independiente de su sistema operativo.

3.2.3 Electron

Electron es un Framework implementado por encima del Chromium, principalmente sirve para empaquetar las aplicaciones webs, así obteniendo una aplicación que sea posible de ejecutar en todos los computadores independiente del sistema operativo, ya sea Windows, macOS o Linux. Usamos este Framework por encima del Angular, así obteniendo una aplicación ejecutable en un computador.

3.2.4 Conclusión: una aplicación híbrida

El objetivo final de este TFG es realizar un monedero de Bitcoin que se puede ejecutar en todos los dispositivos, ya sea dispositivos móviles con sistema operativo Android o iOS, o computadores con sistema operativo Windows, Linux o macOS.

La arquitectura que hemos diseñado consiste en implementar una aplicación web con el Framework Angular como el base del proyecto, y por encima de Angular creamos dos capas del mismo nivel, una es el uso de Framework Ionic para la ejecución en dispositivos móviles, y la otra es el uso de Framework Electron para la ejecución en los computadores.

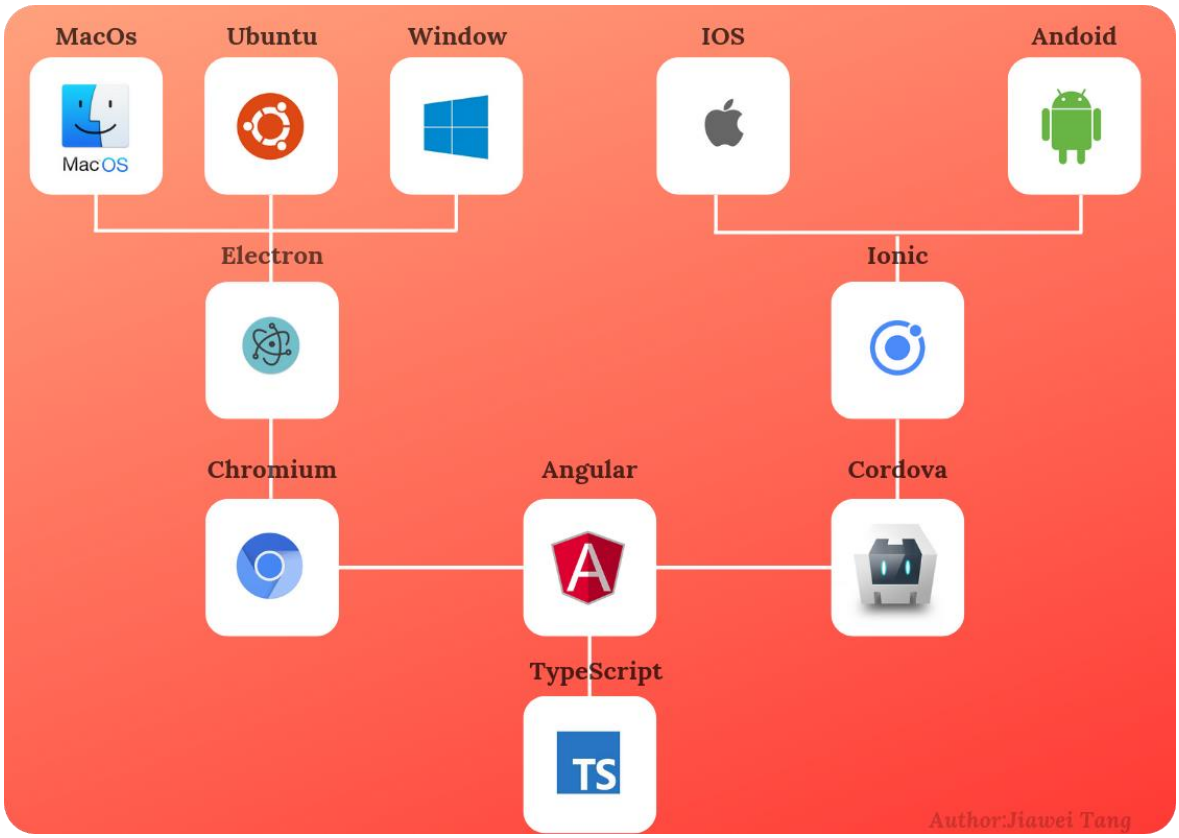


Figura 6. Arquitectura de la aplicación

3.3 Lógica de un monedero ligero

Para implementar un monedero ligero por vía API, es necesario obtener informaciones de *Blockchain* desde los nodos situados en algún servidor (2.1Monedero Bitcoin).

Hay dos opciones para realizar esto:

-Opción 1: crear nuestro propio servidor privado. Para ello es necesario, desplegar un nodo de Bitcoin que contiene todos los datos del Blockchain en un ordenador y tratar el ordenador como un servidor privado. Existe otras formas más simples, que consiste en adquirir un servidor remoto como Amazon Web Services o Microsoft Azure y desplegar el nodo de Bitcoin en el servidor remoto. Una vez desplegado el nodo hay que implementar la lógica del servidor para realizar la comunicación con el cliente. En este caso es recomendable usar el protocolo de comunicación API-REST para recibir las peticiones del cliente (enviar Bitcoin, obtener información de Blockchain etc…), también es necesario implementar un Web Socket para notificar las recepciones de Bitcoins.

-Opción2: usar nodos o servidores públicos y seguros, consiste en obtener informaciones de Blockchain a través de los nodos o servidores públicos y accesibles, sitios como: *URL 1.Servidor público blockchain.com* o *URL 2.Servidor público smartbit.com*. Para realizar la comunicación con estos nodos es necesario usar el protocolo de comunicación API-REST, y Web Socket para las notificaciones de las recepciones de Bitcoin.

www.blockchain.com

URL 1.Servidor público blockchain.com

www.smartbit.com.au/

URL 2.Servidor público smartbit.com

Hemos adoptado la lógica de la Opción 2. Esta opción es mucho más sencilla y simple de implementar, en este caso solo tenemos que enfocar en la lógica de la implementación del monedero de Bitcoin y no en desarrollo en un sistema de servidor-cliente, ya que esto no es el objetivo final de este TFG. Respecto al usuario final de la aplicación, un servidor público ofrece más confianza que un servidor privado, todas las operaciones y datos son mostradas de forma transparente.

Una vez definida la lógica de implementación, es necesario ahora tomar algunas decisiones:

- Usar el API-REST ofrecido por *URL 1.Servidor público blockchain.com* para red principal.
- Usar el Web Socket ofrecido por *URL 1.Servidor público blockchain.com* para red principal.
- Usar el API-REST ofrecido por *URL 2.Servidor público smartbit.com* para red test.
- Usar el Web Socket ofrecido por *URL 2.Servidor público smartbit.com* para red test.
- Usar la recomendación de fee, ofrecido por *URL 3.Feed recomendado.*

https://Bitcoinfees.earn.com

URL 3.Feed recomendado

- Usar la librería BitcoinJs-lib versión 3.3.2 y bips que ayuda a la implementación de la lógica del monedero.
- Usar la librería Crypto de NodeJs.

URL 1.Servidor público blockchain.com es uno de los servidores más conocidos hasta ahora y ofrece las peticiones de API-REST sin límites y con una limitación de peticiones de Web Socket de hasta 8000 peticiones al día. Ofrece los datos de Blockchain de forma transparente y rápida, estos son los motivos principales por el cuál hemos seleccionado este servidor para la red principal de Bitcoin, permite que la aplicación funcione en tiempo real, sin preocuparse de la monitorización en la parte de servidor. En este TFG se va a utilizar *URL 2.Servidor público smartbit.com* para la red test de Bitcoin, el uso de este servidor principalmente es porque no hay limitaciones de peticiones, y esto resulta más fácil y cómodo para el desarrollo.

BitcoinJs-lib es una de las librerías más conocidas y potentes para implementar los monederos de Bitcoin, es usado por grandes compañías como blockchain.com, Exodus, etc … El principal uso de esta librería es facilitar y simplificar nuestro desarrollo y hacer que nuestro código tenga una estructura modularizada y limpia.

La librería Crypto nos ofrece un API para realizar todos los procesos de encriptado, desencriptado y otras operaciones criptográficas.

3.4 Diseño del monedero

Como hemos explicado en la sección *3.2.1 Angular6*, tiene tres partes: los componentes, los servidores y los modelos. Esta sección se va a centrar en el diseño de los servidores (controladores), es aquí donde se concentra toda la lógica que hay por debajo del monedero de Bitcoin. Es posible utilizar TypeScript como un lenguaje orientado a objetos, por lo tanto, es posible en realizar diagrama de clase y usar patrones de diseño.

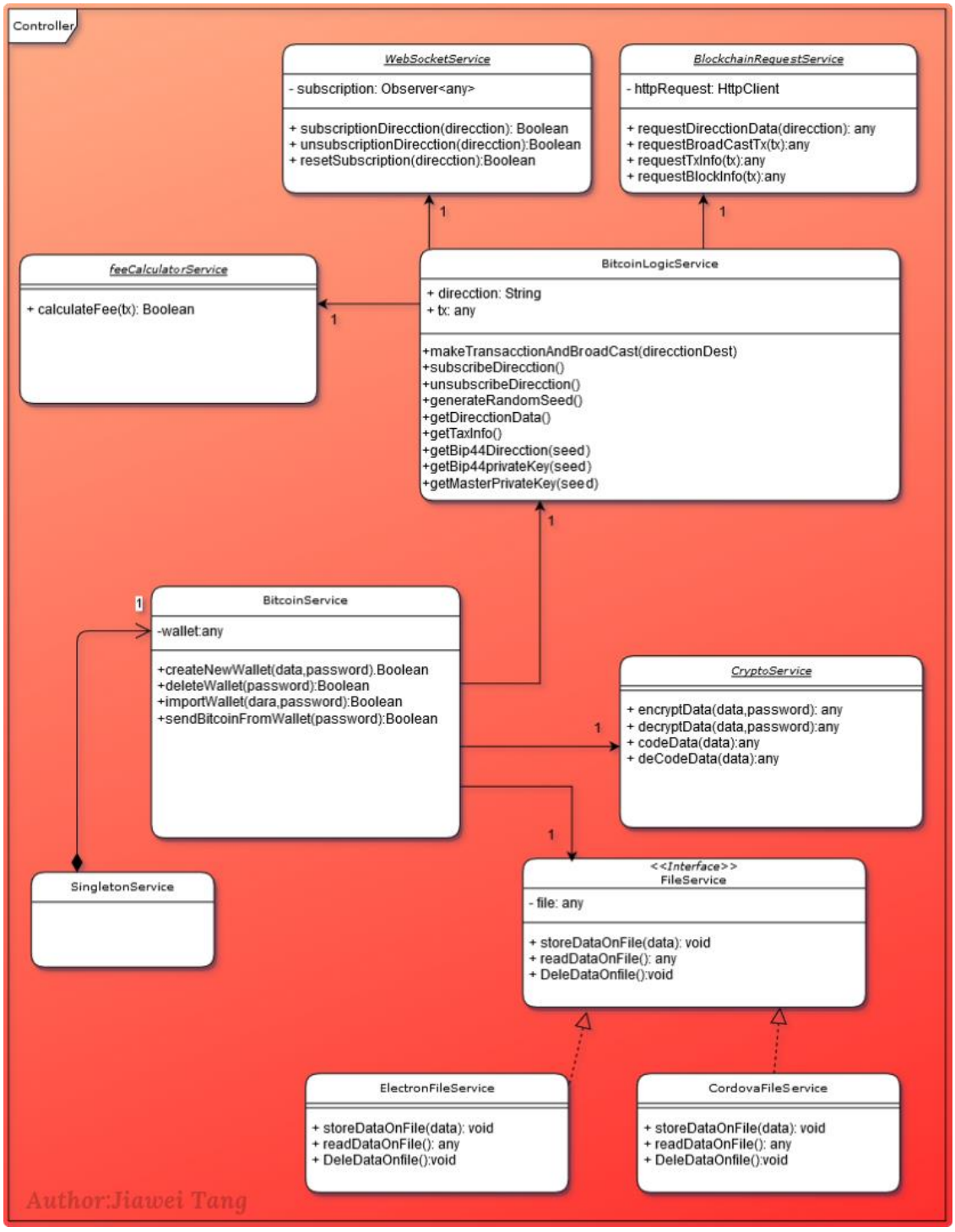


Figura 7.Diseño de la aplicación

Hemos adaptado el uso del patrón de diseño Singleton en este proyecto. Este patrón de diseño permite, el proyecto sea extensible y al mismo tiempo dejar las clases de forma ordenada y modularizada. Aunque ahora mismo dentro de la clase Singleton (*SingletonService*) solo contiene un objeto *BitcoinService* pero resulta útil si en un futuro queremos extender nuestro proyecto, por ejemplo: queremos añadir una nueva criptomoneda *Ethereum* a nuestro monedero. En este caso podemos realizar una extensión muy simple gracias a la clase Singleton, podemos añadir un nuevo servidor que se llame *EthereumService* y la clase Singleton realizará la instancia de esta nueva clase, la implementación de *EtherumService* se realizará de forma aislada sin depender de *BitcoinService.*

Hemos creado varias **clases estáticas** que realiza tareas específicas:

- *WebSocketService*: se encarga principalmente en realizar la conexión de web socket con un servidor público de esta forma recibir todos los movimientos de una dirección de Bitcoin. (*4.5 Recepción de Bitcoin*)

- *CryptoService*: se encarga de realizar el proceso de cifrar, descifrar, codificar y descodificar de los datos. (*4.3 Almacenamiento de la semilla y direcciones*)

- *BlockchainRequestService:* se encarga de realizar todas las peticiones de API-REST para obtener los datos de un servidor público. (*4.4 Datos de una dirección, UTXO*)

- *FeeCalculatorService:* se encarga de calcular el fee estimado de una transacción. (*4.6.2 Fee recomendado*)

Los restos de clases usan las clases estáticas para completar sus funcionalidades:

- *BitcoinService*: se encarga principalmente en la administración del monedero, funcionalidades como: crear monedero, borrar monedero, importar monedero y realizar envío de Bitcoin. Esta clase contiene *BitcoinLogicService* y *CryptoService.*

- *BitcoinLogicServe*: esta es la clase fundamental para este proyecto, en ella se deposita toda la lógica del monedero, funcionalidades en bajo nivel como: generar la clave privada y construir una transacción, en esta clase contienes clases WebScoketService, *BlockchainRequestService* y *FeeCalculatorService.* De esta forma a través de esta clase también podemos realizar las peticiones al servidor y calcular el fee de la transacción. *(4.2 Generación de la clave privada, semilla, 4.6.1 Generación de una transacción y Broadcast)*

Por último, hemos creado una interfaz *FileService* que está diseñada para realizar la escritura y la lectura de los ficheros, dependiendo de la plataforma vamos a implementar de una forma u otra. (*4.3 Almacenamiento de la semilla y direcciones*)

4 Desarrollo

4.1 Introducción

El contenido del desarrollo está modularizado en cinco partes: generación de la clave, almacenamiento de la clave privada, recepción de Bitcoin, envío de Bitcoin y cálculo de fee. Cada uno de estos módulos vendrá algunos ejemplos para facilitar el entendimiento del desarrollo.

4.2 Generación de la clave privada, semilla

La generación de la clave privada es el proceso más importante en un monedero de Bitcoin, este proceso se debe realizar todo desde local sin depender de ningún servidor.

En este proyecto vamos a usar algunos generados de números aleatorios (*1.4.3 Clave Privada, Clave pública, Dirección*), CSPRNG (*Cryptographically Secure Pseudo-Random Number Generator*), este tipo de generador produce datos impredecibles que necesitan por motivos de seguridad. Algunas librerías conocidas que usan este tipo de generador son:

Librería	**Método**	**Source**
Crypto	randomBytes(nBytes)	https://nodejs.org/api/crypto.html
Uuid	uuidv4(Object)	https://www.npmjs.com/package/uuid
Nanoid	nanoid(nLength)	https://www.npmjs.com/package/nanoid

Tabla 2. Librerias y metodos para generar bytes aleatorios

Los métodos mencionados en la *Tabla 2. Librerias y metodos para generar bytes aleatorios* generan los números aleatorios impredecibles y seguros. Hemos utilizado los métodos *randomBytes* y *uuidv4*. La lógica de implementación consiste en obtener un número aleatorio entre 1 y 0, en caso de 0 generamos 16 bytes aleatorios con el método *randomBytes*, en caso de 1 generamos los 16 bytes con el método *uuidv4*. El motivo de esta implementación es para que la generación del número aleatorio no dependa de una sola librería.

Una vez generado los bytes aleatorios, tenemos que convertir estos en la clave privada siguiendo los estándares mencionados en *2.2 BIP*. Necesitamos traducir nuestros bytes generados a hexadecimal, y luego traducirlo a las palabras de diccionario (*BIP0039*), para realizar este proceso se puede usar el método *entropyToMemonic* de la librería *bip39*. Una vez generada la semilla es necesario generar las claves privadas y las direcciones siguiendo el estándar *BIP0032* y el estándar *BIP0044*. Para generar estos datos es necesario utilizar la librería BitcoinJs-lib y escribir la lógica de implementación. Mostraremos en la siguiente sección un pequeño ejemple, que genera una clave privada y dirección con las librerías Bitcoinjs-lib, bip39 y crypto.

```
/*importar la libreria bitcoinJs-lib */
var bitcoin = require('bitcoinjs-lib');
/*importar BIP0039 */
const bip39 = require('bip39');
/*importart crypto */
const crypto = require('crypto');
/*generar randomByte de 16 bytes*/
var randomBytes = crypto.randomBytes(16);
/*cambiar el byte a hexadecimal, y pasar a palabras de diccionario*/
var seed = bip39.entropyToMnemonic(randomBytes.toString('HEX'));
/*obtener el nodo padre del arbol de claves privadas, seed*/
var nodePadreSeed =
bitcoin.HDNode.fromSeedBuffer(bip39.mnemonicToSeed(seed),
bitcoin.networks.testnet);
/* obtener la clave privada correspondiente al indice 0 de la rama 44,
BIP0044*/
var privateKeyWif = nodePadreSeed.derivePath('m/' + '44').keyPair.toWIF();
/* obtener las direcciones correspondientes al indice 0 de la rama 44,
BIP0044*/
var addressBip0044 = nodePadreSeed.derivePath('m/' +
'44').keyPair.getAddress();

/*imprimir los resultados */
console.log(seed.toString('latin'));
console.log(privateKeyWif);
console.log(addressBip0044);
```

Ejemplo 1.Generación de la clave privada

4.3 Almacenamiento de la semilla y direcciones

Una vez generado la clave privada, el contenido de este se tiene que guardar en local, de forma segura y eficiente para que el usuario pueda firmar las transacciones. En esta sección explicamos el algoritmo de encriptación que hemos usado para mantener la clave privada y los métodos de almacenamiento seguro.

La forma más adecuada y estándar para mantener los datos, es depositar los datos en un fichero en formato JSON, y que los datos almacenados se encuentren encriptados y codificados.

Se ha adaptado el uso de algoritmo de encriptación AES256-CTR, este algoritmo es revisado por La Agencia de Seguridad Nacional de los Estados Unidos (NSA) y declaró que todos ellos eran suficientemente seguros para su empleo en información no clasificada del gobierno de los Estados Unidos. Es uno de los algoritmos más seguros y potentes entre todos los que existen, consiste en encriptar los datos con una contraseña de un usuario (más carácter más seguro), tal que, cada vez que el usuario quiere acceder a los datos encriptados se deben desencriptarlos con la contraseña correspondiente.

Mantener los datos encriptados en un fichero es una buena idea, pero los datos encriptados los muestran, de forma transparente para los usuarios, para mantener la privacidad de los datos, vamos a convertir los datos en hexadecimal y luego codificarlos en UTF16-LE.

```
{
    "walletName": "monederoBitcoin",
    "seed": "here is my private key that is the seed are twelve words"
}
```

Ejemplo 2.Fichero1, semilla sin encriptar

```
1
bf7af0993caa29c673880c34f54ba146365b4237ac8967bacff27014af2e9de3a73f1d9
59cd024825bc180c1956906f32ad955646ddebca68824b87a5a7a0c17fa723efdf17eaa
7eb38ac7e7ef86733fd9900040ac86536512e13b43541eb957a28358232b486b5c9bf2d
39a37e11ae
```

Ejemplo 3.Fichero1 clave privada encriptado por contraseña “thisIsMyPasswords123” en formato hexadecimal.

```
戲寠晡您ﾝﾙ 慣㊇ 挹嬛伶十11点 昴仔慢参ﾍ%ﾙ 黶御ﾌﾝ 愷龍坠户捡晦嬴  愴㊅ 愚敷思ﾍっ ㄴㄷ
```

```
愁孜擄㊐ 帆匾换曩捏恆垢廿昶㊍ 撼忉垢坐摀托慣帆㈣ 戴夔叡寙ち | 昷寙ﾌﾝ 晲晤嬪慪寙
```

```
托

伶捡攻攻幩嬛ﾌﾐ 撵悚〰 仜慌龍黶坐ㄥ敲ﾋﾙ 御匾ㄴ托忉愷帇匾㈣㊍ 御堊叡憃晢搹恬9点

攻ㄱ骔
```

Ejemplo 4.Fichero1 clave privada encriptado por contraseña “thisIsMyPasswords123” en formato hexadecimal y codificado en utf16-le

Mantener solamente la semilla en el fichero es bastante cómodo, ya que con este dato podemos generar de nuevo las direcciones y las claves privadas necesarias para firmar una transacción. Existe un pequeño problema de usabilidad en este mecanismo de almacenamiento, si un usuario decide realizar una firma a una transacción, se tiene que usar su contraseña para desencriptar el archivo, leer la semilla contenida en el archivo, generar la clave privada y firmar la transacción. Pero, qué pasa si el usuario solamente quiere ver la cantidad de Bitcoins que tiene en el monedero (es decir la cantidad de Bitcoins que hay en la dirección del usuario) y no quiere realizar ninguna transacción. En este caso el usuario tiene que usar la contraseña otra vez, desencriptar el fichero y obtener las direcciones a partir de la semilla y luego con esta dirección obtener la cantidad de Bitcoins que hay, esto es bastante incómodo para el usuario.

La resolución del problema de usabilidad mencionado anteriormente consiste en crear un segundo archivo que guarda las direcciones del usuario, este archivo no estará encriptado pero sí codificado. Las direcciones de un usuario son públicas por lo tanto no tienen la necesidad de encriptarse, esta solución mejora la usabilidad de la aplicación. El usuario solamente usará la contraseña para realizar firmas de transacciones, y para restos de movimientos leerá el fichero de las direcciones que no se requiere una contraseña.

```
{
    "walletName": "monederoBitcoin",
    "address": [
        "mtXWDB6k5yC5v7TcwKZHB89SUp85yCKshy"
    ]
}
```

Ejemplo 5.Fichero2, dirección sin encriptar

```
2268656 c6c6f22
```

Ejemplo 6.Fichero2 direcciones sin encriptar en formato hexadecimal.

```
㈲帆鱷捐捐昶㈲
```

Ejemplo 7.Fichero2 direcciones en formato hexadecimal y codificado en utf16-le

Los dos ficheros tienen los mismos nombres, pero con diferentes sufijos, para mantener la privacidad de los archivos, los nombres de los archivos están codificados con BASE64.

Fichero	Contenido	Encriptado	Codificación Nombre fichero	Codificación contenido fichero	Nombre del fichero	Sufijo
Fichero1	Nombre del monedero y semilla	AES256-CTR	BASE64	UTF16-LE	Nombre del monedero	Terminado en ".seed"
Fichero2	Nombre del monedero y direcciones	No	BASE64	UTF16-LE	Nombre del monedero	Terminado en ".addr"

Tabla 3.Resumen de los ficheros

4.4 Datos de una dirección, UTXO

En el sistema Bitcoin no existe el concepto de usuarios sino direcciones. El propietario de una dirección es la persona que tiene las claves privadas de esas direcciones. Para obtener los datos de las direcciones usamos el protocolo de comunicación API-REST con servidor público: Smarbit (*3.3 Lógica de un monedero ligero*).

https://testnet.smartbit.com.au/api

URL 4.Documentación API Smartbit

API-REST es una forma de realizar peticiones al servidor con GET, DELETE, POST para obtener, borrar o publicar datos. Vamos a mostrar un ejemplo que realiza una petición GET al servidor Smarbit que obtiene los datos de UTXO de la dirección *msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU:*

https://testnet-api.smartbit.com.au/v1/blockchain/address/msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU/unspent

URL 5.Datos de la dirección msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU

```
{
    "success": true,
    "paging": {
        "valid_sort": [
            "id"
        ],
        "limit": 10,
        "sort": "id",
        "dir": "desc",
        "prev": null,
        "next": null,
        "prev_link": null,
        "next_link": null
    },
    "unspent": [{
        "addresses": [
            "msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU"
        ],
        "value": "0.55000000",
        "value_int": 55000000,
        "txid":
"ee30f332f4c866828da2a88ee52719553126b2f7c74777e93aee4af617b3bf55",
        "n": 0,
        "script_pub_key": {
            "asm": "OP_DUP OP_HASH160
8335caa61c4585eac4d6d01eeed0008457c64b50 OP_EQUALVERIFY OP_CHECKSIG",
            "hex": "76a9148335caa61c4585eac4d6d01eeed0008457c64b5088ac"
        },
        "req_sigs": 1,
        "type": "pubkeyhash",
        "confirmations": 58737,
        "id": 90237144
    }]
}
```


Ejemplo 8.Respuesta API REST

Fijándonos en el *Ejemplo 8.Respuesta API REST*, el campo "unspent" son UTXO de la dirección (*1.4.4.2 Input, Output, Fee*), es decir Bitcoins no gastadas de la dirección *msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU.* "Unspent" es un array, cada elemento de su array hay varios atributos, hay que enfocar solamente en los campos que son relevantes:

- "value" son Bitcoins no gastadas por el usuario.
- "n" es un valor relevante para hacer transacciones, es el índice del Output (cuando un usuario envía Bitcoins a una dirección, es decir crear una nueva transacción, puede generar muchos Outputs, y estos Outputs se identifica con este valor "n". En el ejemplo mostrado en la *Figura 3.Input, Output, sin Fee* , la dirección A n es igual a 0 y la dirección B n es igual a 1).
- "confirmations" son confirmaciones que ha tenido la transacción.
- "txid" es el hash de la transacción.

En este ejemplo el tamaño del array de "unspend" es 1, pero perfectamente puede tener más de una transacción en una dirección, y esto causa que la dirección tenga muchos "unspend" (el tamaño de array de "unspend" en este caso sería mayor que 1) y que la suma de todos los "unspend" es el número de Bitcoins que poseen la dirección. En este ejemplo solo se ha mostrado un tipo de petición al servidor, para más tipos de peticiones hay que consultar en la documentación de la página oficial de Smarbit.

4.5 Recepción de Bitcoin

En la sección *4.5 Datos de una dirección, UTXO*, se ha explicado cómo obtener los datos de una dirección usando API-REST, pero el uso de API-REST no es suficiente para completar todos los requisitos de un monedero de Bitcoin. API-REST es un método de comunicación bajo demanda, en que el cliente realiza una petición y el servidor responde la petición con un dato determinado. El problema surge cuando una dirección envía unos Bitcoins a otra dirección, la dirección receptora para ver las recepciones de Bitcoins es necesario pedir estos datos al servidor vía API-REST, pero cómo sabe el receptor cuando enviar la petición.

El problema planteado anteriormente se soluciona con el uso de Web Socket. Este mecanismo permite la comunicación bidireccional entre el cliente y servidor, el funcionamiento de este es muy simple, el cliente subscribe a un canal del servidor indicando la dirección. El servidor puede enviar mensajes al cliente subscrito sin haber recibido ninguna petición. Esto es bastante útil cuando una dirección recibe Bitcoin en un tiempo determinado y el servidor manda un mensaje al cliente para notificar la llegada de estos Bitcoin.

Hemos adaptado el uso de Web Socket proporcionado por SmartBit, el usuario se debe subscribir *URL 6.web socket smartbit* y mandar un mensaje indicando la dirección de Bitcoin que quiere subscribir. Para realizar la prueba se recomienda usar algún cliente de Web Socket (Browser Websocket Client).

wss://testnet-ws.smartbit.com.au/v1/blockchain

URL 6.web socket smartbit

```
{
    "type": "address",
    "address": "mrQoR4BMyZWyAZfHF4NuqRmkVtp87AqsUh"
}
```

Ejemplo 9.Mensaje de websocket, subscripción de una dirección

```
{
    "type": "subscribe-response",
    "payload": {
      "success": true,
      "message": "Successfully subscribed to address
mrQoR4BMyZWyAZfHF4NuqRmkVtp87AqsUh"
    }
  }
```

Ejemplo 10.mensaje de respuesta de webSocket, tras subscribir en la dirección.

4.6 Envío de Bitcoin

Este capítulo está dividido en dos partes, la primera parte explicamos la formación de una transacción en local usando la librería BitcoinJs-Lib, hasta su difusión en toda la red de Bitcoin usando el servidor de Smartbit. En la segunda parte explicamos la obtención de fee recomendado para las transacciones usando un servidor externo.

4.6.1 Generación de una transacción y Broadcast

Una vez obtenido los UTXO de una dirección (*4.4 Datos de una dirección*), es posible hacer la transacción de Bitcoin en local sin ninguna conexión de Internet, para crear las transacciones usamos la librería BitcoinJs-Lib.

Siguiendo la lógica de la sección *1.4.4.2 Input, Output, Fee*, una transacción consiste en obtener los UTXO de la dirección de envío (Bitcoins no gastadas), añadir estos UTXO como input, y añadir las direcciones de recepción como output, por el último, se realiza la firma de la transacción. Una vez firmada la transacción hay que realizar el proceso de broadcast (difusión de la transacción en la red de Bitcoin). Usamos la conexión de API-REST con Smartbit para completar este proceso.

```
/*importar la libreria bitcoinJs-lib */
var bitcoin = require('bitcoinjs-lib');

/*crear la clave privada en formato WIF que vamos a firmar la transacción
*/
var clavePrivada =
bitcoin.ECPair.fromWIF("cRoMMJLSTiXqanRaSfztQ5sBvgpDqf5R6KaTFrHUnC8mW4F
uFUsJ",bitcoin.networks.testnet);

/*crear una transacción */
var transaccion = new
bitcoin.TransactionBuilder(bitcoin.networks.testnet);

/*suponemos que el valor n es 1 */
var indiceUTOX_n = 1;

/*cantidad de Satoshi Total en la transaccion 96...37c */
var bitcoinTotal = 154500000;

/*cantidad de Satoshi a enviar */
var bitcoinEnviar = 4000000;

/*cantidad de fee en Satoshi*/
var fee = 500000;

/*añadir el TxId de UTXO como input*/
transaccion.addInput("963ff5ceb67d6de42d6c69ba1aabfa93b6bc61426f145170f
99a7545bdaae37c", indiceUTOX_n);

/*añadir la dirección de recepción con 400000 satoshi*/
transaccion.addOutput("msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU",
bitcoinEnviar);

/*añadir la dirección de cambio y 150000000 */
transaccion.addOutput("mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh", bitcoinTotal
- bitcoinEnviar - fee);

/*firmar la transacción */
transaccion.sign(0, clavePrivada);

/*imprimir la transacción firmada */
console.log(transaccion.build().toHex());
```

Ejemplo 11.Desarrollo de una transacción

En el *Ejemplo 11.Desarrollo de una transacción* consiste en crear una transacción con la dirección de envío *mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh,* los datos de las direcciones se han obtenido por vía API (*4.4 Datos de una dirección, UTXO*) en este URL:

https://testnet-api.smartbit.com.au/v1/blockchain/address/mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh/unspent

URL 7.Datos de la dirección *mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh*

La dirección *mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh* hay 1.545 Bitcoins en UTXO, dentro de la transacción *963ff5ceb67d6de42d6c69ba1aabfa93b6bc61426f145170f99a7545bdaae37c* con índice n es igual a 1, queremos enviar 0.04 Bitcoins a la dirección *msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU,* y con una fee de 0.005 Bitcoins. Por lo tanto, hay que crear un input con el hash de la transacción *963ff5ceb67d6de42d6c69ba1aabfa93b6bc61426f145170f99a7545bdaae37c,* y crear dos outputs, el primer output hay que apuntar a la dirección de recepción *msUjFgwj5iQGHc9HsKSFLmtMyS3i2u3NBU* con 0.04 Bitcoins y el segundo output corresponde el Bitcoin que no quieres enviar, es decir el cambio, en este caso son 1.5 Bitcoins (este valor se ha obtenido restando el Bitcoin total, entre Bitcoins enviados y el fee) a la dirección *mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh* (*1.4.4.2 Input, Output, Fee*).

En el *Ejemplo 11.Desarrollo de una transacción* muestra un caso muy simple con solo un input y dos outputs, el sistema de envío se complica cuando la dirección tiene más de un input, si accedemos en la URL 7.Datos de la dirección *mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh* vemos que la dirección *mjY9NJoqpMEzNZ6BFnV1fBxKAQULYcXMhh* tiene 4 UTXO en 4 transacciones (la suma de estos 4 UTXO es el Bitcoin total que poseen la dirección, en este caso 4.978 Bitcoins). Si ahora queremos enviar un valor superior de 1.545 Bitcoins, no hay más remedio de buscar la suma de los UTXOs que superan a 1.545 (podemos coger el UTXO de la transacción *963ff5ceb67d6de42d6c69ba1aabfa93b6bc61426f145170f99a7545bdaae37c* como un input y coger *e7eae078cb047c094a1ccbaa631bbd6c472f5fa12bcc91fb3eddb1fbf94d9609* como el segundo input, con estos dos ya podemos enviar un máximo de 3.878 Bitcoins).

4.6.2 Fee recomendado

Dentro de una transacción fee determina el tiempo que puede tardar una transacción enviada hasta su confirmación en un bloque. El valor de fee es volátil no es buena idea poner un valor constante o un valor aleatorio, hay que buscar una forma de obtener fee recomendada en el momento cuando se realiza la transacción.

El valor recomendado de fee se puede obtener en un servidor externo, *URL 8.Servidor externo para obtener Fee recomendado.* Este servidor ofrece el fee recomendado en Satoshi por Bytes.

https://Bitcoinfees.earn.com/api/v1/fees/recommended

URL 8.Servidor externo para obtener Fee recomendado

```
{
    "fastestFee": 18,
    "halfHourFee": 18,
    "hourFee": 12
}
```

Ejemplo 12.Ejemplo fee recomendado por vía API REST

El uso de este servidor para obtener fee es bastante cómodo, pero hay un problema dentro de este sistema. El API nos ofrece fee recomendado en Satoshi por Bytes de una transacción, pero no sabemos cuántos bytes tiene la transacción, si la transacción se construye con la fee ya calculada.

Para resolver el problema hay que usar una librería llamada “coinselect”. Esta librería es capaz de calcular un fee estimado con los datos de input, output y fee recomendado de Satoshi por bytes.

```
/*importar la librería coinSelect */
const coinSelect = require('coinselect');
/*fee recomendado obtenido por api*/
const feeBytePorSatoshi = 18;
/* declarar utxos*/
var utxos = [
    {
        txId: '...',
        vout: 0,
        value: 10000
    }
];
/*declarar los inputs*/
var targets = [
    {
        address: '...',
        value: 5000
    }
];
/*usar el método coinselect para calcular el fee */
var {inputs, outputs, fee } = coinSelect(utxos, targets, feeBytePorSatoshi);

/*imprimir fee estimado*/
console.log(fee);
```

Ejemplo 13.Cálculo de fee recomendado

5 Pruebas unitarias y resultados

En este capítulo se encuentra los resultados obtenidos de las pruebas unitarias y resultados en ejecutar la aplicación en los diferentes entornos: Android y Windows.

5.1 Pruebas unitarias

Las pruebas han sido realizas en el siguiente entorno:

Sistema Operativo: Windows 10 pro 64bit
Memoria Ram: 16GB
Procesador: Intel Core i5-7600K @ 3.86GHz

Para realizar las pruebas unitarias de cada módulo, hemos usado algunos Framework de prueba: Jasmine y Karma.

Jasmine es un Framework de código abierto para realizar pruebas unitarias de JavaScript. Karma es una herramienta que nos permite generar las pruebas en el navegador, y ver los resultados en el mismo navegador.

La prueba consiste en instanciar la clase de prueba, creando unos datos ficticios como parámetros de entrada a los métodos de pruebas. Se debe comprobar si la salida del método es lo esperado. En caso positivo significa que la prueba se ha realizado correctamente, en caso negativo significa, error en la implementación del método.

5.1.1 Resultado de pruebas unitarias

En esta sección se va a mostrar resultados obtenidos, en realizar pruebas unitarias a todos los módulos.

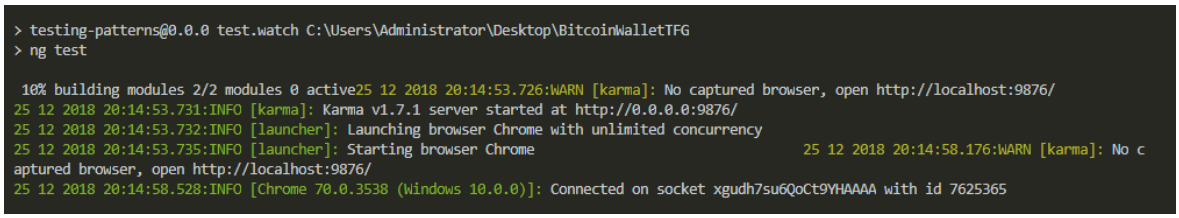


Figura 8.Entorno de la prueba unitaria

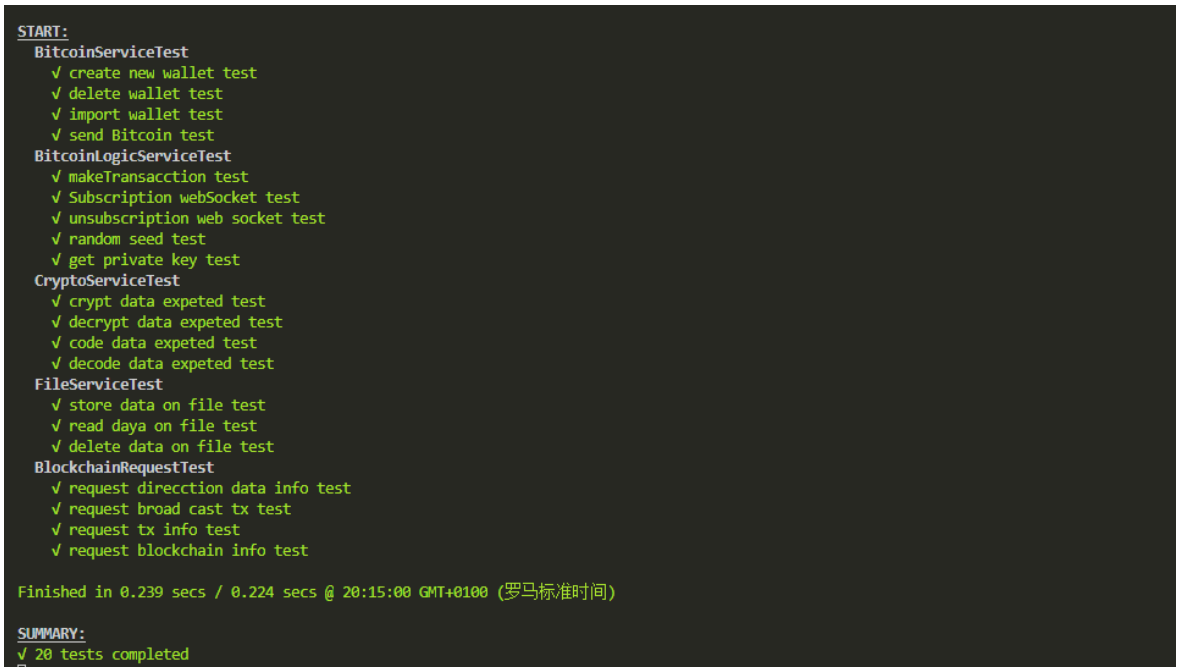


Figura 9.Salida de la prueba unitaria_1

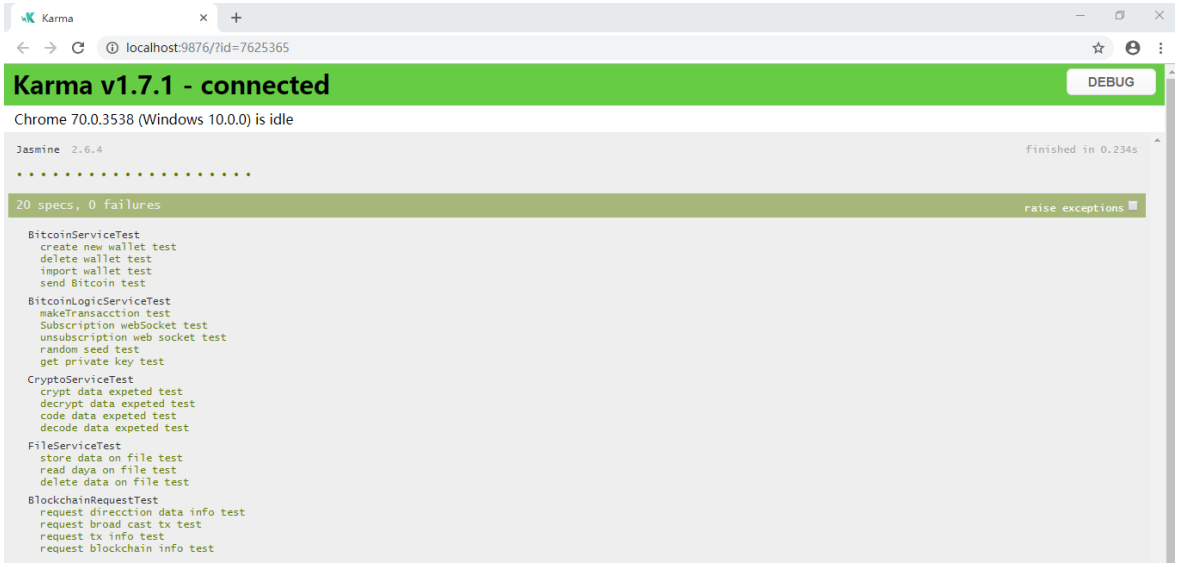


Figura 10.Salida de la prueba unitaria_2 consola Karma

Resultado obtenido de las pruebas:

Pruebas totales realizadas: 20
Errores de la prueba: 0
Tiempo de prueba: 0.224 segundos
Porcentaje de error en la prueba: 0%
Porcentaje de corrección en la prueba: 100%

5.2 Resultados de ejecución en sistema Operativo Android

La ejecución ha sido realiza en el siguiente entorno:

Dispositivo: Honor 8 FRD-L09
Sistema operativo: EMUI 5.0.3
Versión de Android: 7.0
Memoria Ram: 4GB
Procesador: Hisilicon Kirin 950

Figura 14. Panel crear monedero nuevo

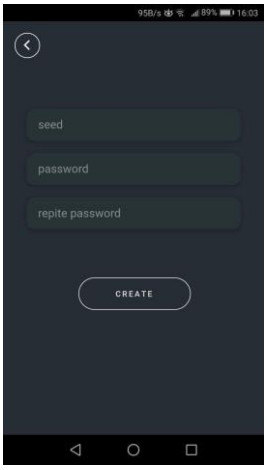


Figura 12.Panel formulario importar nuevo monedero

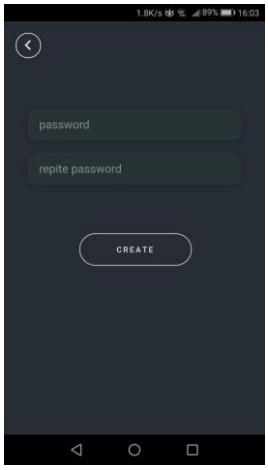


Figura 13.Panel formulario crear nuevo monedero

Figura 11. Panel principal de la aplicación

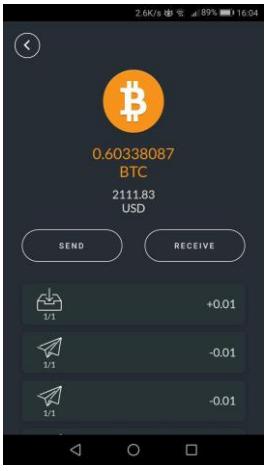


Figura 17 Panel historial de transacción.

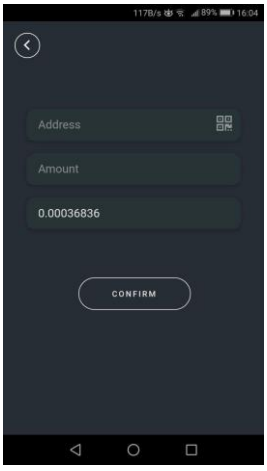


Figura 16.Panel formulario enviar Bitcoin

Figura 15.Panel QR recibir Bitcoin

La aplicación se ha podido ejecutar correctamente en una máquina de Android sin errores. Las capturas mostradas no contienen todos los paneles de la aplicación, sino solamente las fundamentales para mostrar la ejecución correcta de la aplicación.

5.3 Resultados de ejecución en sistema Operativo Windows

La ejecución ha sido realiza en el siguiente entorno:

Sistema Operativo: Windows 10 pro 64bit
Memoria Ram: 16GB
Procesador: Intel Core i5-7600K @ 3.86GHz

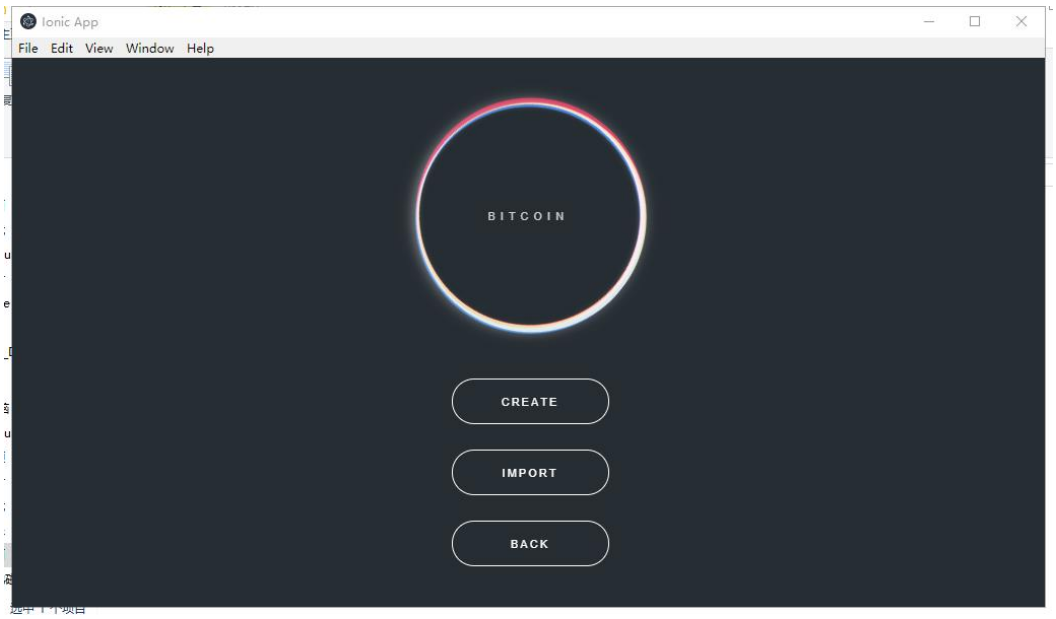


Figura 18.Panel crear monedero nuevo Windows

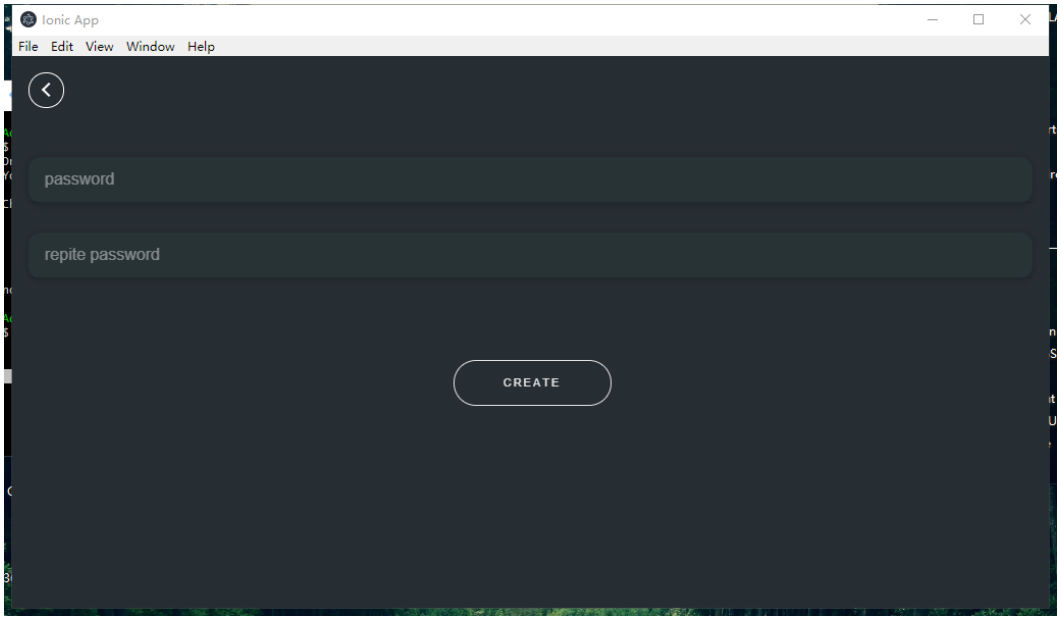


Figura 19.Panel formulario crear nuevo monedero Windows

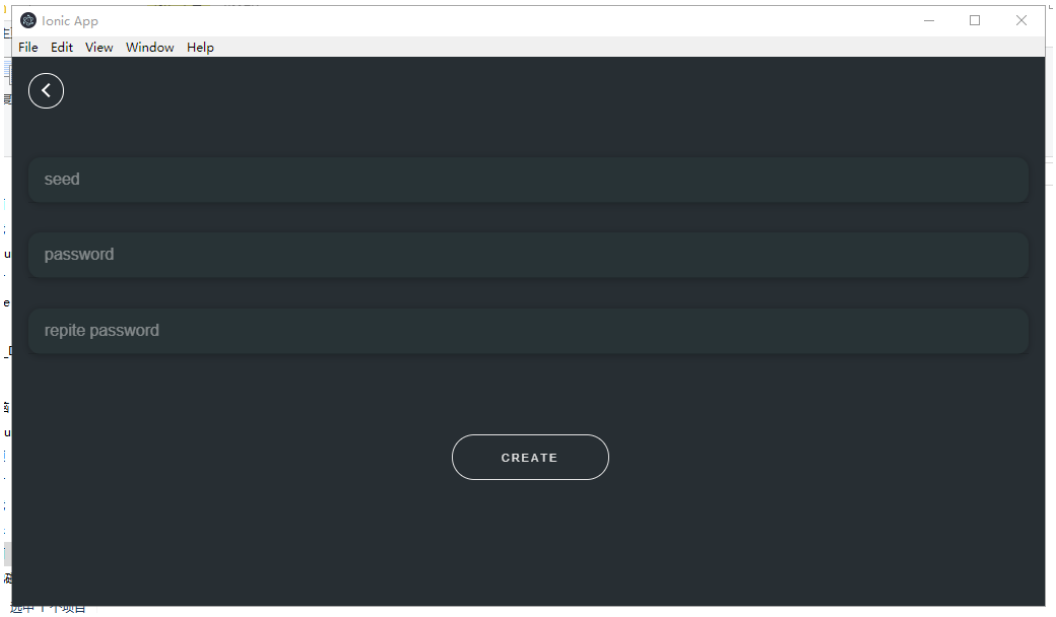


Figura 20.Panel formulario importar nuevo monedero Windows

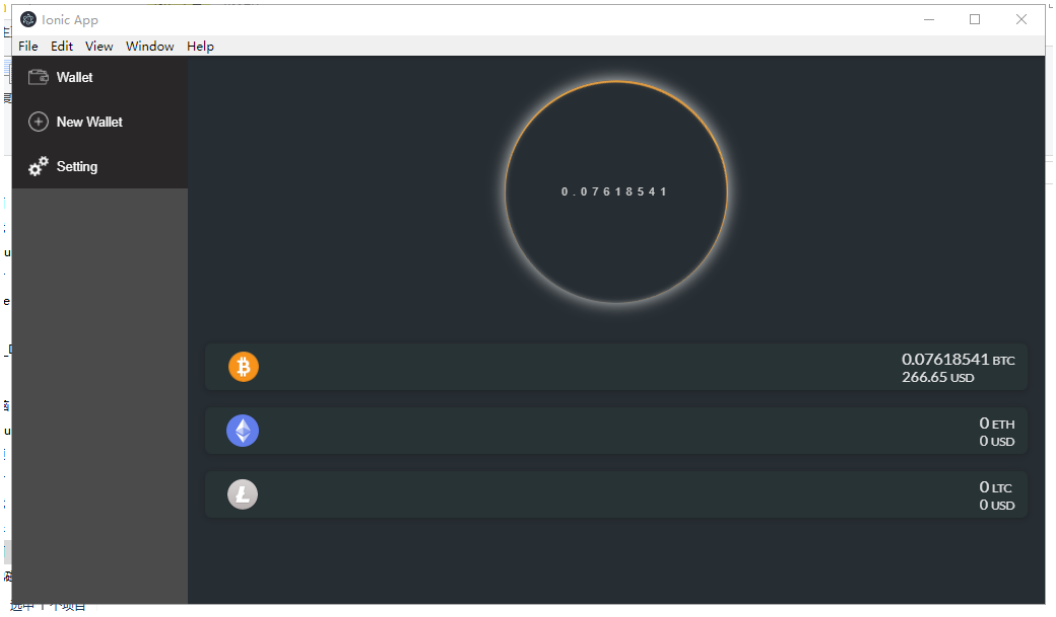


Figura 21.Panel principal de la aplicación Windows

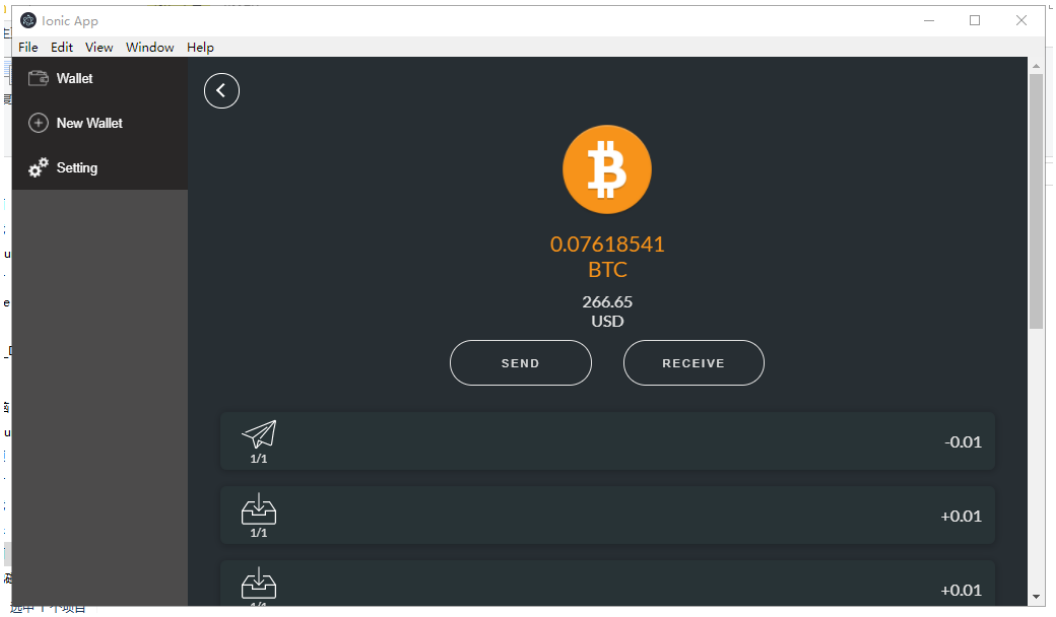


Figura 22.Panel historial de transacción Windows

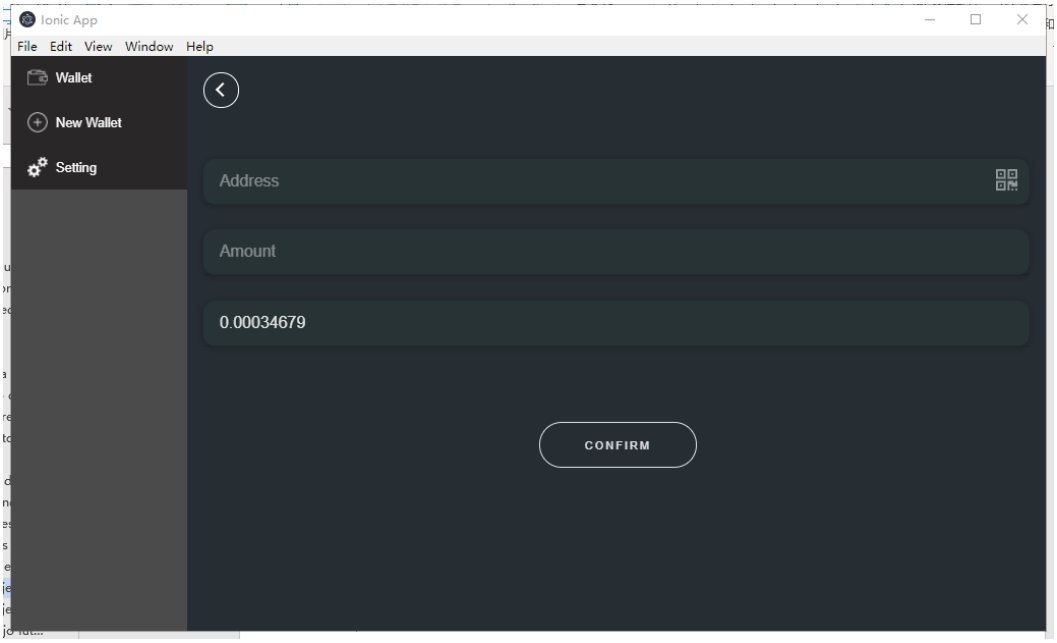


Figura 23.Panel formulario enviar Bitcoin Windows

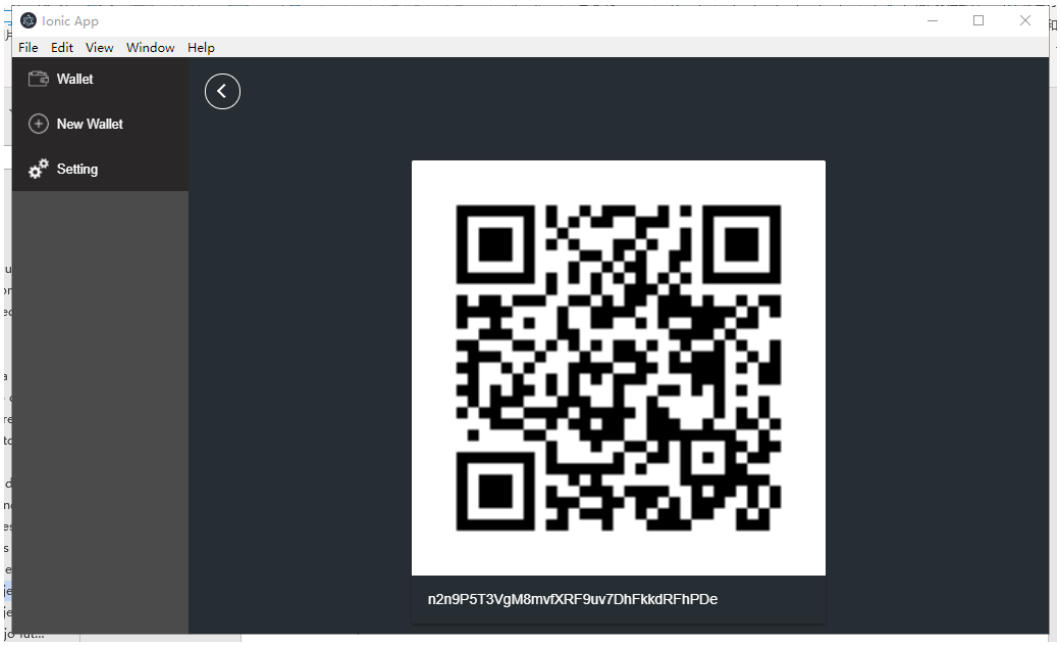


Figura 24.Panel QR recibir Bitcoin Windows

La aplicación se ha podido ejecutar correctamente en una máquina con un sistema operativo Windows sin errores. Las capturas mostradas no contienen todos los paneles de la aplicación, sino solamente las fundamentales para mostrar la ejecución correcta de la aplicación.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Este TFG partió de la idea de crear un monedero de Bitcoin ligero determinista, cumpliendo algunos requisitos fundamentales: depositar Bitcoin, crear nuevo monedero, almacenar Bitcoin de forma segura, enviar Bitcoin y recibir Bitcoin.

Durante el desarrollo de este TFG, se ha adquirido fundamentos y conceptos importantes de Bitcoin, y además de eso, se ha dado mucha importancia en el diseño de cómo va a ser nuestro monedero. Hemos tomado la decisión de crear una aplicación hibrida que es posible ejecutar en todas las plataformas independientes del sistema operativo. Crear este tipo de aplicación ha sido verdaderamente un reto para nosotros, lo más importante de este proceso ha sido la toma de decisión de las librerías, Framework y protocolos que hemos usado en este TFG. Hemos seleccionado Angular6, Ionic4 y Electron para diseñar nuestra aplicación, el uso de algunas librerías como BitcoinJs-Lib, CryptoJs y los protocolos de comunicación API-REST y Web Socket.

En la parte de codificación, podemos decir que es la parte más divertida de este TFG. En ella se ha usado todos los conceptos de Bitcoin y hemos programado las operaciones que podemos hacer con Bitcoin en bajo nivel, por otro lado, hay que dar la importancia a los conocimientos de criptografías y codificaciones que hemos usado para proteger los datos del usuario. El uso de Frameworks y los patrones de diseño en este trabajo nos ha podido simplificar y modularizar el código, y poder realizar fácilmente el debugeo, en caso de error.

Dejando de lado los resultados, también valoro los conocimientos adquiridos. Sin duda, me ha servido de aprendizaje de cómo desarrollar una aplicación hibrida, obviamente la parte más importante no es el desarrollo de la aplicación sino entender Bitcoin, el concepto, la lógica y el funcionamiento de este elemento.

6.2 Trabajo futuro

A pesar de que en general los resultados han sido satisfactorios, aún hay mucho margen de mejora, más en algunas partes que en otras. Estos serían los principales aspectos a mejorar en un futuro. Extender el monedero que soporte varias criptomonedas como: Ethereum, Litecoin, BitcoinCash etc…, añadir más algoritmos de cifrados, ahora mismo solo soporta el algoritmo de cifrado AES-256-CTR y por último la mejora de la interfaz de usuario.

Referencias

[1] Mastering Bitcoin: Unlocking Digital Cryptocurrencies by Andreas M. Antonopoulos, December 27th 2014 by O'Reilly Media, ISBN 1449374042 (ISBN13: 9781449374044)

[2] Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World, Published May 10th 2016 by Portfolio (first published 2016), ISBN 0670069973 (ISBN13: 9780670069972)

[3] BitcoinJs-Lib open source doc "https://github.com/Bitcoinjs/Bitcoinjs-lib"

[4] Wikipedia of Bitcoin "https://en.Bitcoin.it/wiki/Main_Page"

[5] Angular documentation "https://angular.io/docs", Ionic documentation "https://ionicframework.com/docs/", Electron documentation "https://electronjs.org/docs"

Glosario

AES256-CTR	Algoritmo de cifrado con 256 bits, cuando más potente es la contraseña, más seguro son los datos cifrados.
Apache Cordova	Entorno desarrollo de aplicaciones móviles, útil para ejecutar códigos HTML, CSS y JavaScript en cualquier dispositivo móvil.
API-REST	Protocolo de comunicación de la red, es el más usado hoy en día consiste en establecer una comunicación entre cliente y servidor bajo demanda, mediando peticiones de recursos con operaciones: GET, POST, PUT, DELET
Base58	Algoritmo de codificación, muy usado en el sistema de Bitcoin consiste en codificar los datos en rango de caracteres A-Z, a-z y 0-9 omitiendo los caracteres: “0”, “O”, “I”, “l”, “+”, /
Base58 CheckSum	Digito de 4 bytes que se suma a la dirección de Bitcoin antes de codificarlos en Base58, para asegurar el correcto formato de la dirección de Bitcoin. Estos dígitos se calculan realizando doble SHA256 hash de los datos PrimerosCuatroBytes(SHA256(SHA256(datos))).
Base64	Algoritmo de codificación, consiste en codificar los datos en rango de caracteres A-Z, a-z, 0-9, “/” y “+”.
Chromium	Proyecto de navegador web de código abierto de Google. Es un navegador completamente funcional por sí solo y proporciona la mayor parte del código para el navegador Google Chrome. Su motor de navegación “V8” es usado por Electron para desarrollar aplicaciones.
ECC	Elliptic curve cryptography, variante de la criptografía asimétrica o clave pública basada en las matemáticas de las curvas elípticas
Framework	Es una estructura en capas que indica qué tipo de programas pueden o deben construirse y cómo se interrelacionan.
Hash160	Hash obtenido por realizar SHA256 Y RIP160 de los datos, HASH160 = RIPMD160(SHA256(datos))
HD walled	Hierarchical Deterministic wallet, monedero de bitcoin determinista que cumple el estándar de BIP0032, estos tipos de monederos se caracterizan por tener muchas claves privadas y direcciones, generadas desde una única semilla.

JSON	Un estándar establecido para la comunicación de datos en la red, también es tratado como un objeto en JavaScript. Los datos que se comunicación con API-REST es de formato JSON.
MVC	Patrón de diseño, modelo, vista y controlador.
P2P	Red distribuida donde todos participantes tiene la misma importancia jerárquica.
RIPMED160	RACE Integrity Primitives Evaluation Message Digest, Función hash de 160 bits.
RSA	Sistema criptográfico de clave pública, es válido tanto para cifrar como para firmar digitalmente.
Satoshi	Unidad en que se mite Bitcoin, 1 Satoshi = 0.00000001 BTC.
SHA256	Secure Hash Algorithm, Función hash de 256 bits
Singleton	Patrón de diseño, donde los objetos se instancian solo una vez dentro de una sola clase.
UTF16-LE	Forma de codificación de caracteres Unicode utilizando símbolos de longitud variable.
UTXO	Bitcoins no gastados.
Web-Socket	Protocolo de comunicación de la red, permite una comunicación bidireccional entre el servidor y cliente.
WIF	Wallet Import Format, formato de codificación de las claves privadas de Bitcoin para que sea más corta y estándar. Es el resultado obtenido tras codificar los datos con Base58.

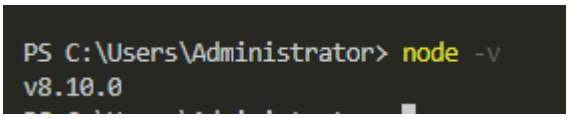
Manual para ejecutar los ejemplos

Para ejecutar los ejemplos de prueba:

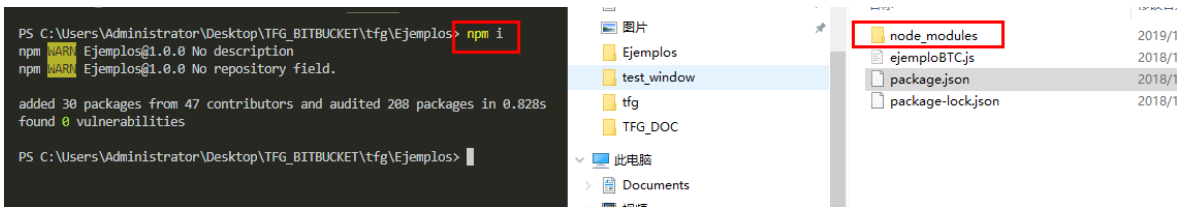
1º Crear un fichero llamado "*package.json*", con los siguientes contenidos

```
{
  "name": "Ejemplos",
  "version": "1.0.0",
  "description": "",
  "main": "ejemploBTC.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@types/Bitcoinjs-lib": "^3.4.0",
    "bip39": "^2.5.0",
    "Bitcoinjs-lib": "^3.3.2"
  }
}
```

2º Comprobar la instalación de nodeJs con "**node -v**", si no está instalado hay que descargarlo en este enlace https://nodejs.org/en/



3º Mover el archivo "*package.json*" a una carpeta, e instalar los paquetes con el comando "npm i"



Este comando descarga todas las dependencias y genera una carpeta llamada *node_modules*, y un archivo *package-lock.json*

4º Crear un archivo llamado "*ejemploBTC.js*" y copiamos el ejemplo de prueba. Para ejecutar hay que usar el comando "node ejemploBTC.js"